Model Checking Verification and Validation at JPL and the NASA Fairmont IV&V Facility[1] -

Frank Schneider, Jet Propulsion Laboratory, California Institute of Technology, Steve Easterbrook, NASA IV&V Facility, Jack Callahan and Todd Montgomery, West Virginia University
Contact: Francis.L.Schneider@jpl.nasa.gov

Abstract

We show how a technology transfer effort was carried out. The successful use of model checking on a pilot JPL flight project demonstrates the usefulness and the efficacy of the approach. The pilot project was used to model a complex spacecraft controller. Software design and implementation validation were carried out successfully. To suggest future applications we also show how the implementation validation step can be automated. The effort was followed by the formal introduction of the modeling technique as a part of the JPL Quality Assurance process.

Introduction

Following the pilot use of model checking at NASA JPL and the IV&V Facility [1], and at NASA Ames[2], we have followed five steps in introducing model checking to the Quality Engineering process at JPL. First, references [1] and [2] show model checking to be an effective tool in validating the behavior of spacecraft systems. Second, our model checking results were then carried forward to validate the software implementation for the presence of design anomalies. Third, having validated the implementation by hand, we show how the process can be automated. Fourth, we have documented the process to be used in a development environment by incorporating and generalizing the above elements. Finally, we are engaged in applying the methodology developed here on future spacecraft.

Model Checking as a Validation Tool

We use model checking to mean the process of (1) abstracting a partial specification from requirements and design elements for a reactive system and (2) applying reachability analysis to the resulting partial specification to validate that it has properties of interest. A reactive system is one that takes input from its environment at unpredictable times and responds according to a specific set of rules. We have previously shown model checking to be an effective tool in validating the behavior of a fault tolerant embedded spacecraft controller [1]. That case study shows that by judiciously abstracting away extraneous complexity, the state space of the model could be exhaustively searched allowing critical functional requirements to be validated down to the design level. The system we validated was a two-fold redundant spacecraft controller. It consists of a prime system that controls the spacecraft bus and a backup system. The backup system receives synchronization information from the prime system via the spacecraft bus. The purpose of the system is two fold. First, it has to respond to and repair must-fix-spacecraft faults.

[1] The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Second, it must complete execution of high priority sequences. To realize these goals two mechanisms were utilized.

First, the system uses a checkpointing scheme that allows:

- Execution to be frozen when a fault occurs
- Repair of the fault somewhere in the spacecraft
- Rollback to the start of the last incomplete subsequence
- Resumption of sequence execution

Accordingly, the checkpointing scheme allows efficient sequence execution since completed subsequences need not and in many cases can not be repeated. The checkpointing scheme requires three seconds of aging for each new checkpoint before the new checkpoint is considered to have been encountered. This is caused by fault leakage detection time such that a fault at the end of a previous subtask may not be detected until up to three seconds after the beginning of a new subtask. This could mean that the fault precluded instructions at the end of the previous subtask from being executed.

Second, the overall redundancy of the system made up of prime and backup controllers allows the entire prime controller to fail. Failure is detected by the backup system that then becomes prime and takes over execution where the failed system halted. The backup system becomes prime; takes over control of the spacecraft bus; completes repairing the fault; rolls back to the start of the last incomplete subsequence and resumes execution of the sequence. Figure 1 illustrates the architecture involved. Further details can be found in reference [1].

The initial abstracted design state space contained about 2^{87} states. By this statement we mean that we estimate there to be 2^{87} different combinations of variable values and conditions that completely describe every possible configuration of the spacecraft controller. There are five types of faults such that the controller is required to respond to one type of fault at a time. Because fault detection and recovery requirements could be handled one-at-a time, the requirements were partitioned into five equivalence classes accordingly reducing the state space to be searched significantly. The state space was further reduced by removing states from the finite state machine representation that did not contribute to the checkpointing scheme we were attempting to validate. This gave rise to a new estimate of about 100, 000 states. The resulting Harel Chart [3] for the abstracted spacecraft controller is that shown in Figure 1.

Example: Sequence execution segment:

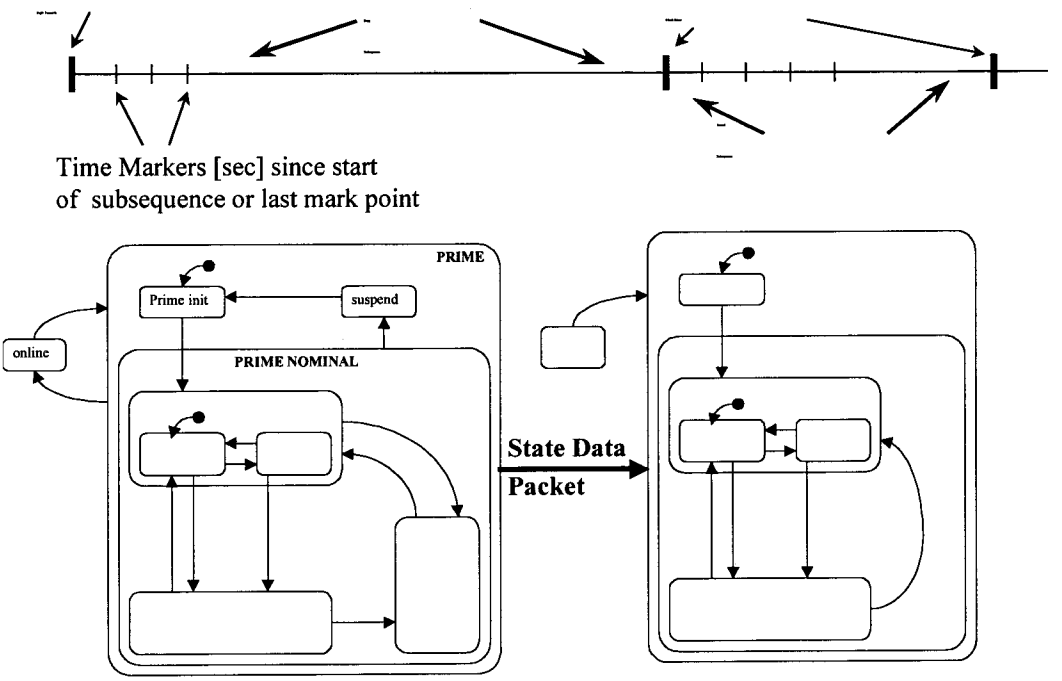

Figure 1

The validation was accomplished with the SPIN model checking system [4]. Six separate rollback requirements on the rollback scheme were validated. Three anomalies were uncovered with the model checker traversing about 130,000 states for each anomaly with run time being approximately 30 seconds for each anomaly.

Anomaly one resulted from repeated prime failure causing loss of synchronization with the backup system. This result occurs when the prime system experiences repeated intermittent failures possibly due to the same fault, and such that the prime system repairs the fault in less than one second. According to our model this would mean that notice of the fault would never be propagated to the backup system. Consequently, the backup system could get significantly ahead of the prime system in the execution of its own copy of the sequence. Then should the prime system subsequently fail, the backup system could roll back to an incorrect location. This anomaly is due to the ordering of processing described in the requirements specification.

Anomaly two depends on how faults are handled at the end of the sequence. Should a fault occurrence be detected up to within three seconds of execution of the last instruction, there would be no rollback after repair of the fault. This is the case since the last instruction in the sequence was not identified as a checkpoint. However, should a fault occur prior to the end of

the sequence, according to the fault leakage detection rule there is no guarantee that all instructions at the end of the sequence would have been successfully executed. Our validation run failed because our model assumed that once the sequence completed, the backup and the prime systems returned to the Power Up Idle state; accordingly, there would be no sequence to return to once the fault was corrected. This anomaly is due to a missing requirement.

The third anomaly concerns the occurrence of a fault 2 seconds after a checkpoint is encountered in the prime string. The prime string freezes its aging function at n + 2 seconds. Since faults that occurred in the previous second are not broadcast to the backup system until the current second it will continue to execute, aging its checkpoint by one further second. At this point the backup system receives notice of the fault and freezes its aging process. However, it now has an erroneous rollback point. Should the prime system subsequently fail, the backup system would roll back to an incorrect address. This requirement is an error in the detailed requirements. This is so since the error would not go away by making the checkpoint-aging buffer shallower or deeper. It would just make the anomaly occur at a different location.

Software Implementation Validation

We have subsequently validated the implementation for the presence of the three design anomalies. For this purpose we used a special purpose spacecraft simulator called the High Speed Simulator (HSS) [5, 6]. The simulator uses code identical to the real spacecraft. However, it is de-coupled from hardware and telemetry. Accordingly, its use as a test vehicle (1) is an accurate measure of system functionality and (2) it allows rapid turnaround on test suite creation, execution, and reporting of results.

The simulator allows test engineers to write test sequences for execution on the simulator. Given the data structures present in the spacecraft controller, a Tool command language (Tcl) program is written that orchestrates (1) the execution of the test sequence, (2) the extraction and printing of values of selected data attributes (3) the extraction and printing of any relevant time stamps and (4) fault injection scenarios and their responses.

1.1 Procedural Steps

We wanted to know if the software implementation contained the same anomalies as were found in the design. To determine this, we supplied the High Speed Simulator with a simple sequence program for execution. By injecting faults into the running sequence, the same problematic conditions would be set up in the implementation that were discovered by design validation. Our earlier validation work derived the design anomalies from a three-step process. First, the prime system would stop running freezing its check point ager in response to a fault occurrence somewhere in the spacecraft. Second, the prime system would load and begin execution of a fault recovery program. Finally, during its execution of the fault recovery program, the prime system itself would fail. To affect this same scenario in the software implementation, the prime system was commanded to do a cold boot at execution points in the implementation identical to those that caused the anomalies in the design validation. An operational backup system considers the prime

system cold boot to be a prime system failure. It reacts by becoming prime itself; taking control of the spacecraft bus; rolling back to the relevant earlier check point address if necessary; and resuming execution of the sequence program. For example, the third anomaly found in the design validation process occurs when the prime system fails after encountering a fault scenario that freezes its check point at second two in the aging process. This results in the new prime system rolling back to an inappropriate address due to a timing problem in the design. Accordingly, cold booting the prime system when it has aged its checkpoint by two seconds has the same effect as the two step process considered in the design case.

Detection of the presence of design anomalies in the implementation was done by selecting data structures for output identical to those used in the design case. These output data values taken together at any execution cycle represent the state of the implementation at a particular point in time. As the implementation executes, this 'state vector' describes a finite state machine that represents the implementation. This finite state machine is an abstracted finite state machine since it doesn't include all variables, only the ones considered relevant to the current validation. If a corresponding design anomaly is itself present in the implementation, the implementations' abstracted state vector will go through an equivalent sequence to that found in the design validation done earlier. In this case the work proceeded by outputting each state vector for the executing implementation. The output list was then manually examined line by line to look for the presence of anomaly states.

The input sequence program that was incorporated into the HSS Tcl interface program to check for the presence of anomalies in the implementation is shown in Figure 2.

```
IP      Mnemonic
800     BEGIN
803     NOP
805     NOP
807     NOP
809     NOP
80b     NOP
80d     CHECKPOINT
80f     NOP
811     NOP
813     NOP
815     NOP
817     NOP
819     CHECKPOINT
81b     NOP
81d     NOP
81f     NOP
821     NOP
823     NOP
825     END
```

Figure 2 Sequence Validation Program

To keep the analysis as straight forward as possible, each instruction was executed on one-second boundaries. A HSS Tcl interface program was written to generate the output state vector sequence of the abstracted implementation state machine. Schematically, the overall process is shown in Figure 3.

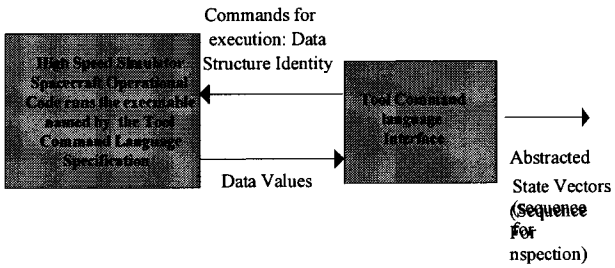

Figure 3: Implementation Abstracted State Machine

The implementation was validated at this point by simply looking at the results of the simulation by hand and recognizing that a design anomaly was or was not reproduced in the output. This means visually examining the output sequence labeled "Abstracted State Vectors" to check the rollback process functionality. Two of the three anomalies found in the design validation were present in the implementation. A brief summary of the results follows.

Implementation Anomaly Validation Results

The first anomaly resulted from repetitive errors that caused the prime and the backup system to get out of synchronization. Our design anomaly fault scenario required a series of prime-fault-repair sequences each of one-second duration or less. We did not see the first anomaly in the system. Further investigation with system engineers revealed that all faults take at least several minutes to repair. Therefore, repair time was extended so that anomaly one would not be seen.

The second anomaly occurs when a fault occurs less than three seconds after the sequence ends. In this case, there is no rollback. That is, once the sequence has been completed there is no rollback in response to an error injected inside the three-second-rollback window. Therefore, there is no guarantee that all instructions at the end of the sequence would have been carried out by the spacecraft. Accordingly, on this basis, the last instruction in the program should have been identified as a rollback point. Our technique demonstrated that the second anomaly was present in the implementation.

The third anomaly results from a fault that brings the prime system down when its aging buffer contains a check point rollback address that has been aged by two seconds. According to our model checking validation, this information would not get to the backup system until the following

second, thereby causing its two deep backup buffer to age its rollback address by an additional second. Consequently, its rollback address would be consistent with a three-second delay following a checkpoint when only two seconds had elapsed since the prime string had executed its last instruction. Prime system failure was again caused by cold booting the prime string at the point it had aged its checkpoint by two seconds. The subsequent rollback in the new prime system did not match the old prime's rollback address. Accordingly, our technique demonstrated that the third anomaly was present in the implementation.

The cold boot process is equivalent to the injection of a single fault that brings the prime system down. This process causes the overall spacecraft controller to fail to conform to requirements since control in the new prime system rolls back to an inappropriate location. Therefore, our technique also demonstrated that the overall system made up of prime and backup systems was not single fault tolerant.

All of these results were taken with respect to the spacecraft software as it existed on the High Speed Simulator.

Automating the Validation Process

Dillon and Ramakrishna show how test oracles can be generated from linear temporal logic specifications [7]. Log files generated from a running implementation can then drive these automata. The log files generated are used to drive requirements automata into accepting states should strings from the language they accept be traversed and output by the implementation. The automata are usually specified to check for requirements violations. Using these ideas, we have extended our work on design verification and validation [1] and applied it to the validation of the generic spacecraft controller's implementation. Our results used the output of the running spacecraft simulator system. The real time output was used to drive the automaton that represents one of the anomalies found by model checking. The resultant system was then made up of the spacecraft simulator; the test scenario generator, and automaton representing the requirement to be tested. The result system, called the Automated Validation System (AVS) did detect a counter example in the output indicating the presence of the design anomaly in the implementation. Additionally, the automaton has the capability to output the state vector trail taken by the implementation as it encountered the anomaly thereby giving information on how the anomaly develops as execution proceeds.

We have proposed that this concept be used as a fault protection mechanism on autonomous spacecraft. These spacecraft have self sufficient activities based on a set of high-level mission objectives carried on board the spacecraft. See for example [10]. The AVS would provide an effective and robust fault detection and response system for such spacecraft. The steps to be followed are outlined below.

1. Intercept the autonomously developed activity or action routine that the spacecraft is to carry out based upon and derived from the current mission profile.
2. Parse the mission profile or its more detailed on-board-generated requirements and derive from them the logical condition that represents the requirements that are to hold during and at termination of the executing action routine.
3. Express the logical condition in the linear temporal logic (LTL).

4. Define any macros that may be necessary to map the derived LTL automaton into any required ancillary form.
5. Produce the executable fault detection automaton from the LTL formula derived from steps 3 and 4.
6. Annotate the action routine so that it outputs an abstracted state vector representing an essential model of the action routine.
7. Couple the output from step 6 to the executable automaton produced in step 5.
8. Execute the overall action routine piping its real time output to the LTL automaton as it is produced. A fault condition will then drive the LTL automaton into one of its accepting states indicating that the associated requirement has been violated. When this condition is detected, respond autonomously to the fault. If this is not feasible, notify ground control and begin to safe the spacecraft as may be apropos of the situation.

1.2 Critique of Recommendations

The production of the executable LTL automaton cited in step 5 need not be a complex stumbling block. For example, safety conditions on total available power, maximum turn angles, antenna pointing and the like are easily quantified. The production of an automaton that checks for a liveness condition has already been illustrated in this section for design anomaly three of the spacecraft controller checkpoint process. Additionally, any of several other automata systems could be used depending on analyst choice.

The advantage of the system proposed here is that detailed knowledge of the underlying spacecraft software is not required nor would it ever be necessary.[2] Once the appropriate data structures governing the requirements are located, they can be output to the LTL automaton in the form of an abstracted state vector. Additionally, if the autonomous system makes use of an architecture analogous to the High Speed Simulator architecture, the process would be considerably more straightforward and precise. Here, once the appropriate data structures were identified, they would be easily tagged for inclusion in the abstracted state vector. All of this is again easily an automation step. In this latter case, consideration of the action routine per se would be minimized.

The usefulness of the procedure described here is that the automata theory is well understood, predictable, and easily programmable. Systems engineers would however have somewhat of a learning curve to become proficient in expressing requirements specifications in the linear temporal logic.

It might be argued that because the low level sequence and requirments are derived from high level mission objectives that AVS derived requirements are of course always going to work out. This would be the case should it be proven that the deductive logic used in producing detailed low level commands is valid in all circumstances. Therefore, one might argue what is the point of the procedure at all? However, such an argument would not be valid for it is a physical spacecraft in a physical universe dynamically making decisions about its environment. First, things can and do often go awry aboard the spacecraft. Single Event Upsets, bus failures, cameras jamming, valves freezing up or not opening properly on first try, squibs misfire and a host of others. Second, estimates of relative locations of external objectives can be misjudged. If distance estimates are accurate, angles and velocities can be way off due to low measurement resolution in the region where they are made.

Although the example in the text used a single AVS, the number of threads that might be running scales linearly. Accordingly, many such AVSs could be dynamically created as required and released when no longer necessary. Also, there is no reachability problem here as occurs in model checking due to the on-the-

[2] The Time Rover Company discusses their innovative test system at http://www.time-rover.com/SpecLang.html. It embeds LTL logic in the form of language statements within executable routines. Accordingly, it may be difficult to implement such a system within the context discussed here where on-the-fly validation is required for newly generated procedures.

fly-one-time nature of the problem here. We are in fact only interested in current behavior, not in all possible future behaviors. Therefore an AVS for maximum and minimum angles, power, closest distance of approach, and the like could be easily and dynamically configured for each scenario to be carried out.

Accordingly, the AVS system described here can provide a powerful, fast, reliable, first line of defense towards assuring mission success in the Laboratories' unmanned exploration efforts of the 21st century.

Formalization of the Model Checking Process

Having successfully shown the applicability of the model checking process on a spacecraft system, we decided to formalize the methodology. Our introduction of the modeling technique was via the use of a "process chart." A process chart is a flowchart that details the methodology that is applied to accomplish in our case a quality assurance technique. There are currently 21 processes for which process charts exist. Example process charts include Training, Risk Assessment, Requirements Assessment and Design Assessment. In addition to the process chart, each process also has an accompanying summary that details the contents of the process flow. The purpose of the process charts is to (a) document our own processes (b) to be able to convey in a clear and unambiguous way to our customers what our QA processes are and (c) in cases where customers want further assistance, we give guidance to them on how they can carry out their part of the resulting interface. The model checking verification and validation process chart includes high level guidance on how incremental design modeling is carried out over a project lifecycle for reactive systems and their components. This includes using the results of incremental design modeling to check to see if the design anomalies are present in an implementation. Callahan and Montgomery have discussed the use of this approach adopted here within the context of a model-checking environment in their development of the RMP Protocol [8]. In addition to using model checking to find design anomalies, the implementation is also checked in an incrementally evolving development environment. If a faulty design is subsequently corrected, the condition of the implementation can then be checked to see that it reflects the new design. Conversely, should the partial implementation get ahead of the partial design, then the implementation can be used to check the design. In this way an evolving partial implementation and a partial design can be driven to maintain phase coherence with each other. The process thereby yields an implementation that has a much higher confidence level associated with it. Additional details concerning this approach can be found in [9].

Ongoing Work

A test harness similar to the one we have used on the generic spacecraft validation effort is being constructed for a future series of deep space missions called X2000. Presently X2000 includes missions to Pluto and Europa. We are planning to use the validation methodology described here on the X2000 project.

Summary

We have shown model checking to be a viable and useful technology to apply towards making future spacecraft designs and their corresponding implementations more robust. A method was suggested whereby a validation scheme called the Automated Validation System could be used to provide an analytic framework to wrap conventional fault detection and response mechanisms

aboard autonomous spacecraft. The Verification and Validation process using model checking was formalized at the Laboratory by adding the Model Checking process to our Office 506 Quality Assurance process methodology system. Our effort at applying the technology to future spacecraft is a work in progress. We plan to publish a more detailed analysis of the results given in this paper.

References

[1] Francis L. Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann: Validating Requirements for Fault Tolerant Systems using Model Checking. ICRE 1998: 1-13.

[2] Michael R. Lowry, Klaus Havelund, John R. Penix: Verification and Validation of AI Systems that Control Deep-Space Spacecraft: ISMS 1997: 35-47

[3] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-74, 1987.

[4] G. J. Holzmann, "The Model Checker Spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, 1997.

[5] Reinholtz, WK and Robison, WJ.,III, "TheZIPSIM series of high-performance, high fidelity spacecraft simulators," *Proceedings AIAA/Utah State University Annual Conference on Small Satellites,* Aug 29-sept 1, 1994.

[6] Patel, K and Reinholtz, W and Robison, W, "High-speed simulator: A simulator for all seasons", *Proceedings International Symposium on Space Mission Operations and Ground Data Systems (SPACEOPS96,* Munich, Germany Sept 16-20 1996; pg 749-756

[7] L.K. Dillon and Y.S. Ramakrishna: Generating Oracles From Your Favorite Temporal Logic specifications: SIGSOFT'96 CA, USA 106-117

[8] J. R. Callahan and T. L. Montgomery: An Approach to Verification and Validation of a Reliable Multicasting Protocol: ISSTA 96: 187-194

[9] John Callahan, Francis Schneider, Steve Easterbrook: Automated Testing Using Model-Checking: Invited talk Bellcore division of Bell-Laboratories: 1996 - see also http://swayer/~sch/

[10] N. Muscettola, B. Smith, C. Fry, S. Chien, K. Rajan, G. Rabideau, and D. Yan. "On-Board Planning for New Millennium Deep Space One Autonomy," Proceedings of the IEEE Aerospace conference, Snowmass CO, 1997.